Argument Clinic How-To

リリース *3.10.18*

Guido van Rossum
and the Python development team

7 月 08, 2025

目次

author Larry Hastings

概要

Argument Clinic は CPython の C ファイルのプリプロセッサです。builtin の中の、退屈な引数解析のコードを自動化するのが目的です。このドキュメントでは、C 関数を Argument Clinic 化する方法を示し、さらに高度な Argument Clinic の利用方法について説明します。

現在のところ、Argument Clinic は CPython の内部専用の扱いです。CPython の外にあるファイルはサポートしておらず、将来のバージョンでの後方互換性を保証しません。言い換えると、CPython の外の C 拡張をメンテナンスしている場合、Argument Clinic を試してみることはできますが、次のバージョンの CPython の Argument Clinic では互換性が無くなりそのコードが動かなくなる **可能性があります** 。

1 Argument Clinic の目的

Argument Clinic の第一目標は、CPython の中の全ての引数解析のためのコードを引き継ぐことです。ある関数を Argument Clinic を使うように変更すると、その関数は一切の引数解析を行わなくなります。Argument Clinic が生成するコードがブラックボックスになり、CPython がそのブラックボックスの先頭を呼び出し、最後にその関数のコードが呼ばれます。引数の `PyObject *args` (と `PyObject *kwargs`) は暗黙的に、必要な型の C の変数に変換されます。

Argument Clinic の第一目標を達成するためには、使いやすくなければなりません。現在、CPython の引数解析ライブラリを利用するのは面倒で、とても多くの場所で冗長な情報のメンテナンスを必要とします。Argument Clinic を使うことで DRY を実現できます。

もちろん、新たな問題を持ち込むことなしに、自分の問題を解決してくれるのでなければ、誰も Argument Clinic を使いたいとは思わないでしょう。なので、正しいコードを生成することは Argument Clinic にとって最重要です。生成したコードによって速度が大きく低下することがあってはいけないし、速くなるならより良いです。(最終的には、Argument Clinic は大幅な高速化を可能にするはずです。Argument Clinic が、汎用的な CPython の引数解析ライブラリを呼び出す代わりに、オーダーメイドの引数解析コードを生成するようにできるからです。これにより可能な限り最速の引数解析が可能になるでしょう！)

加えて、Argument Clinic は全ての引数解析の方式に対応できるように柔軟でなければなりません。Python の幾つかの関数はとても特殊な引数の解釈を行います。Argument Clinic の目標はその全てをサポートすることです。

最後に、もともとの Argument Clinic の目的は CPython 組み込み関数に ” シグネチャ” の introspection を提供することでした。今までは、組み込み関数に対して introspection する関数で問い合わせると例外が発生していました。Argument Clinic によってこれは過去の事になりました。

Argument Clinic を使うにあたって 1 つ覚えておかないといけないアイデアがあります: それは「より多くの

情報を与えれば、より良い仕事ができるようになる」ということです。Argument Clinic は今のところはまだ比較的シンプルです。しかし、将来の進化により、与えられた情報を元により洗練された賢いことができるようになるでしょう。

2 基本的な概念と使用法

Argument Clinic は CPython とともに配布されています。その中の `Tools/clinic/clinic.py` を見つけることができるでしょう。そのスクリプトを、引数に C ファイルを指定して実行すると:

```
$ python3 Tools/clinic/clinic.py foo.c
```

Argument Clinic は指定されたファイルをスキャンし、次の行と全く同じ行を探します:

```
/*[clinic input]
```

その行を見つけたら、今度は正確に次のような行を見つけるまで、すべてを読み込みます:

```
[clinic start generated code]*/
```

これらの 2 つの行の間のすべての行が Argument Clinic への入力になります。これらのコメント行の開始と終了も含めたすべての行が、Argument Clinic " ブロック" と呼ばれます。

Argument Clinic はその block をパースすると、出力を生成します。その出力は、その C ソースファイルの該当する block の直後に挿入され、チェックサムを含むコメントで終了します。その結果 Argument Clinick block は次のようになります:

```
/*[clinic input]
... clinic input goes here ...
[clinic start generated code]*/
... clinic output goes here ...
/*[clinic end generated code: checksum=...]*/
```

Argument Clinic を同じファイルに対して 2 度実行した場合、Argument Clinic は古い出力を新しい出力で上書きしてチェックサムも更新します。しかし、入力が変更されていない場合、出力も変化しません。

Argument Clinick block の出力部分を編集してはいけません。出力が望みどおりになるまで、入力を変更してください。(出力部分に行った編集は、次に Argument Clinic が新しい出力を書き込む際に失われてしまいます。チェックサムの目的は出力部分に行った編集がないかどうか検出するためです。)

混乱を避けるために、Argument Clinic で利用する用語を定義しておきます。

- コメントの最初の行 (`/*[clinic input]`) は **スタートライン** です。
- Argument Clinic を実行する前の状態のコメントの最終行 (`[clinic start generated code]*/`) は **エンドライン** です。
- 最後の行 (`/*[clinic end generated code: checksum=...]*/`) は **チェックサムライン** です。
- スタートラインとエンドラインの間が **インプット** です。

- エンドラインとチェックサムラインの間が **アウトプット** です。
- スタートラインからエンドラインまで、その 2 行を含めた全体が **ブロック** です。(Argument Clinic がまだ正常に処理を実行できてないブロックは、アウトプットとチェックサムラインをまだ持っていませんが、それもブロックとして扱います。)

3 関数を変換してみよう

Argument Clinic の動作について把握する一番の方法は、1 つの関数で実際に試してみることです。なのでここでは、1 つの関数で試すための最小限の手順を説明します。CPython のコードにコミットする場合は、あとで出てくるもっと強力な機能 (”return converter” や ”self converter” など) を使った変換をする必要があることに気をつけてください。ここでは学習目的でシンプルな手順だけにとどめます。

飛び込もう！

0. まず CPython の最新版のチェックアウトを用意してください。

1. まだ Argument Clinic を利用していない、`PyArg_ParseTuple()` か `PyArg_ParseTupleAndKeywords()` を呼び出している Python 組み込み関数を見つけてください。この例では `_pickle.Pickler.dump()` を利用します。

2. `PyArg_Parse` 関数が以下のいずれかのフォーマット単位を使っていた場合:

```
O&
O!
es
es#
et
et#
```

   あるいは `PyArg_ParseTuple()` の呼び出しを複数持っていた場合、別の関数を選びましょう。Argument Clinic はこれらすべてのケースを **サポートしています**。ですが、これは高度な話題になります。最初の関数にはシンプルなものを選びましょう。

   また、その関数が、同一の引数が複数の型を持つ場合に対応するために `PyArg_ParseTuple()` か `PyArg_ParseTupleAndKeyword()` の呼び出しを複数持っていたり、PyArg_Parse 関数を利用していないようなら、その関数は Argument Clinic に向いていません。Argument Clinic はジェネリック関数やポリモーフィックな引数をサポートしていません。

3. その関数の上に次の定型文を追加し、ブロックを作ります:

```
/*[clinic input]
[clinic start generated code]*/
```

4. docstring を切り取って `[clinic]` の行の間に貼り付け、適切にクォートされた C の文字列になるようにガラクタを全て削除します。そうすると、80 文字以上の行がない、左マージンに揃えられた、テキストだけになるはずです。(Argument Clinic は docstring 内のインデントを保持します。)

古い docstring の最初の行に関数シグネチャのようなものがある場合は、その行を破棄します。(この古い docstring は最早必要ありません — 将来あなたのビルトインで `help()` を使うときは、古い docstring の最初の行にあったような関数シグネチャのようなモノは、実際の関数のシグネチャに基づいて自動的に生成されます。)

例:

```
/*[clinic input]
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

5. あなたの docstring に「要約」(summary) 行がない場合、Argument Clinic は文句を言います。なので、それがあることを確認しましょう。「要約」行は、docstring の先頭にあり、80 桁以内の単一の行で構成される段落である必要があります。

   (例題に引用した元の docstring は、たまたま要約行のみで構成されていたため、このステップでは例題のコードを変更する必要はありませんでした。)

6. docstring に関数の名前を入力し、その後に空白行を入力します。これは関数の Python 名であり、関数への完全なドット・パス (full dotted path) である必要があります。つまり、それはモジュール名で始まり、サブモジュールが含まれている必要があり、関数がクラスのメソッドである場合は、クラス名も含める必要があります。

   例:

```
/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

7. モジュールまたはクラスがこの C ファイルの Argument Clinic で初めて使用される場合は、モジュール および/または クラスを宣言する必要があります。Argument Clinic 界隈では、これらを C ファイルの先頭近くの別のブロックで宣言することが好ましいとされます。これは、インクルード・ファイルや static が先頭に配置されるのと同一の手法です。(なお、このコード例では、説明の都合上 2 つのブロックを続けて表示しています。)

   クラス名とモジュール名は、Python で表示されるものと同一にする必要があります。`PyModuleDef` または `PyTypeObject` で定義されている名前を適宜確認してください。

   クラスを宣言するときは、C では、その型の 2 つの側面も指定する必要があります。それは、このクラスのインスタンスへのポインタに使用する型宣言と、このクラスの `PyTypeObject` へのポインタに使用する型宣言です。

   例:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
```

(次のページに続く)

(前のページからの続き)

```
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

8. 関数のための各引数 (parameters) を宣言します。各引数は、それぞれ独立した行である必要があります。すべての引数行は、関数名と docstring からインデントさせる必要があります。

   これらの引数行の一般的な形式は以下のとおりです:

   ```
   name_of_parameter: converter
   ```

   引数にデフォルト値がある場合は、コンバーター (converter) の後に追加します:

   ```
   name_of_parameter: converter = default_value
   ```

   「デフォルト値」に対する Argument Clinic のサポートは非常に洗練されています。詳細については、**下記 引数のデフォルト値 セクション** を参照してください。

   引数群の下に空白行を追加します。

   「コンバーター」(converter) は、C で使用される変数の型と、実行時に Python の値を C の値に変換する方法の、両方を確立します。ここでは、「レガシー・コンバーター」(legacy converter) と呼ばれるものを使用します。これは、古いコードを Argument Clinic に簡単に移植できるようにするための便利な構文です。

   各引数について、その引数の「フォーマット単位」を `PyArg_Parse()` のフォーマット引数からコピーし、引用符に囲まれた文字列にして、*that* にコンバーターとして指定します。(「フォーマット単位」(format unit) とは、引数解析関数に変数の型とその変換方法を伝える `format` パラメータの 1 〜 3 文字の部分文字列の正式な名前です。フォーマット単位の詳細については、arg-parsing を参照してください。)

   `z#` のような複数文字のフォーマット単位の場合、2 または 3 文字の文字列全体を使用します。

   例:

   ```
   /*[clinic input]
   module _pickle
   class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
   [clinic start generated code]*/

   /*[clinic input]
   _pickle.Pickler.dump

       obj: 'O'
   ```

(次のページに続く)

(前のページからの続き)

```
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

9. 関数のフォーマット文字列に | が含まれている場合、つまり一部の引数にデフォルト値がある場合は、無視してかまいません。Argument Clinic は、デフォルト値があるかどうかに基づいて、どの引数がオプションなのかを推測します。

   関数のフォーマット文字列に $ が含まれている場合、つまりキーワードのみの引数を取る場合は、最初のキーワードのみの引数の、前の行に * を単独で指定し、引数行と同一のインデントを行います。

   (なお、_pickle.Pickler.dump にはどちらも含まれていないため、サンプルは変更されていません。)

10. 既存の C 関数が、(PyArg_ParseTupleAndKeywords() ではなく、) PyArg_ParseTuple() を呼び出す場合、そのすべての引数は位置のみです (positional-only)。

    Argument Clinic ですべての引数を位置のみ (positional-only) としてマークするには、引数行と同一のインデントにして、最後の引数の次の行に / を追加します。

    現在のところ、これは、すべての引数が位置のみ (positional-only) であるか、全てがそうでないかのどちらかです (all-or-nothing)。(Argument Clinic は将来はこの制限を緩和する可能性もあります。)

    例:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

11. 各引数に引数ごとの docstring (per-parameter docstring) を記述すると助けになります。ただし、引数ごとの docstring はオプションですので、必要に応じて、この手順はスキップできます。

    引数ごとの docstring を追加する方法: 引数ごとの docstring の最初の行は、引数行定義よりもさらにインデントする必要があります。この最初の行の左マージンは、引数ごとの docstring 全体の左マージンを確定します。あなたが書くすべてのテキストは、この左マージン量だけアウトデントされます。必要に応じて、複数行にまたがって、好きなだけテキストを書くことができます。

    例:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
```

(次のページに続く)

```
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

12. ファイルを保存して閉じ、そのファイルに対して `Tools/clinic/clinic.py` を実行します。運が良ければすべてうまくいきます --- つまり、Argument Clinic ブロックに出力があり、かつ、`.c.h` ファイルが生成されまます！ テキスト・エディタでそのファイルを再度開いて確認します:

```
/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

static PyObject *
_pickle_Pickler_dump(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: output=87ecad1261e02ac7 input=552eb1c0f52260d9]*/
```

Argument Clinic が全く出力を生成しなかった場合、それは入力にエラーがあったためです。Argument Clinic が問題なくファイルを処理するまで、エラーの修正と再試行を繰り返します。

読みやすさのために、ほとんどのグルー・コードは `.c.h` ファイルに生成されています。これを元の `.c` ファイルにインクルードする必要があり、通常は clinic モジュール・ブロックの直後に置きます:

```
#include "clinic/_pickle.c.h"
```

13. Argument Clinic が生成した引数解析コードが既存のコードと基本的に同一であることを再確認します。

1 番目に、両方の場所で同じ引数解析関数が使用されていることを確認します。既存のコードは `PyArg_ParseTuple()` または `PyArg_ParseTupleAndKeywords()` のいずれかを呼び出す必要があります。Argument Clinic によって生成されたコードが **まったく同じ** 関数を呼び出すことを確認してください。

2 番目に、`PyArg_ParseTuple()` または `PyArg_ParseTupleAndKeywords()` に渡されるフォーマット文字列は、コロンまたはセミコロン含めて、既存の関数で手書きされたものと **まったく同じ** でなければなりません。

(Argument Clinic は常に : の後に関数名が続くフォーマット文字列を生成します。既存のコードのフォーマット文字列が ; で終わっていた場合のこの変更は使用法のヘルプを提供するためで、この変更は無害です。心配してないでください。)

3 番目に、引数 (parameters) のフォーマット単位が 2 つの arguments (length 変数や、エンコード文字列や、変換関数へのポインタなど) を必要とする場合は、2 番目の argument が 2 つの関数呼び出し間で **正確に同じである** ことを確認してください。

4 番目に、ブロックの出力部分内に、このビルトインの適切な static な `PyMethodDef` 構造体を定義するプリプロセッサ・マクロがあるはずです:

```
#define __PICKLE_PICKLER_DUMP_METHODDEF     \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O, __pickle_Pickler_dump__doc__},
```

この static な構造体は、このビルトインの既存の static な `PyMethodDef` 構造体と **まったく同じ** でなければなりません。

これらの項目のいずれかが **何らか異なる** 場合は、Argument Clinic 関数の仕様を調整し、同一になるまで `Tools/clinic/clinic.py` を再実行し続けてください。

14. その出力の最終行が、あなたの "impl" 関数の宣言であることに注意してください。これは、ビルトインの実装が行われる場所です。あなたが変更中の関数の既存のプロトタイプを削除してください。しかし、開き波括弧 `{` は残してください。そして、その引数をパースするコードと、その引数をダンプするすべての変数の宣言を削除してください。Python の引数がこの impl 関数の引数になっていることに注意してください。実装でこれらの変数に異なる名前が使用されている場合は、修正してください。

   少々奇妙なコードなので、もう一度やってみましょう。あなたのコードは今や以下のようになっているはずです:

```
static return_type
your_function_impl(...)
/*[clinic end generated code: checksum=...]*/
{
...
```

   Argument Clinic は、チェックサム行とそのすぐ上の関数プロトタイプまでを生成しました。あなたは関数の開始の波括弧 `{` (および終了の波括弧 `}`) と、その内側の実装を記述する必要があります。

   例:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/
/*[clinic end generated code: checksum=da39a3ee5e6b4b0d3255bfef95601890afd80709]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
```

(次のページに続く)

```
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

PyDoc_STRVAR(__pickle_Pickler_dump__doc__,
"Write a pickled representation of obj to the open file.\n"
"\n"
...
static PyObject *
_pickle_Pickler_dump_impl(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: checksum=3bd30745bf206a48f8b576a1da3d90f55a0a4187]*/
{
    /* Check whether the Pickler was initialized correctly (issue3664).
       Developers often forget to call __init__() in their subclasses, which
       would trigger a segfault without this check. */
    if (self->write == NULL) {
        PyErr_Format(PicklingError,
                     "Pickler.__init__() was not called by %s.__init__()",
                     Py_TYPE(self)->tp_name);
        return NULL;
    }

    if (_Pickler_ClearBuffer(self) < 0)
        return NULL;

    ...
```

15. この関数の PyMethodDef 構造体を含むマクロについて思い出して下さい。この関数の既存の PyMethodDef 構造体を探して、それをマクロへの参照に置き換えて下さい。(ビルトインがモジュール・スコープにある場合、既存の PyMethodDef 構造体はおそらくファイルの終わり近くにあります。ビルトインがクラス・メソッドの場合、既存の PyMethodDef 構造体はおそらく実装の後ろにありますが、比較的実装に近いです。)

    マクロの本体には末尾にカンマ (comma) が含まれていることに注意してください。したがって、既存の static PyMethodDef 構造体をマクロに置き換える場合は、末尾にカンマ (comma) を追加しないでください。

    例:

```
static struct PyMethodDef Pickler_methods[] = {
    __PICKLE_PICKLER_DUMP_METHODDEF
    __PICKLE_PICKLER_CLEAR_MEMO_METHODDEF
    {NULL, NULL}                /* sentinel */
};
```

16. 回帰テスト (regression-test) のスイートの関連部分をコンパイルして実行します。この変更により、新たな、コンパイル時の警告やエラーが発生するべきではなく、Python の動作に外部から目に見える変化が生じるべきではありません。

Well, except for one difference: `inspect.signature()` run on your function should now provide a valid signature!

Congratulations, you've ported your first function to work with Argument Clinic!

4 高度なトピック

Now that you've had some experience working with Argument Clinic, it's time for some advanced topics.

4.1 シンボルのデフォルト値

The default value you provide for a parameter can't be any arbitrary expression. Currently the following are explicitly supported:

- 数値定数 (整数 (integer) と浮動小数点数 (float))
- 文字列定数
- `True` と `False` と `None`
- 必ずモジュール名で始まる、`sys.maxsize` のような単純な記号定数

In case you're curious, this is implemented in `from_builtin()` in `Lib/inspect.py`.

(将来的には、*CONSTANT - 1*' のような式を表現可能にするために、さらに精巧にする必要があるかもしれません。)

4.2 Argument Clinic が生成した関数と変数をリネームする

Argument Clinic automatically names the functions it generates for you. Occasionally this may cause a problem, if the generated name collides with the name of an existing C function. There's an easy solution: override the names used for the C functions. Just add the keyword `"as"` to your function declaration line, followed by the function name you wish to use. Argument Clinic will use that function name for the base (generated) function, then add `"_impl"` to the end and use that for the name of the impl function.

For example, if we wanted to rename the C function names generated for `pickle.Pickler.dump`, it'd look like this:

```
/*[clinic input]
pickle.Pickler.dump as pickler_dumper

...
```

The base function would now be named `pickler_dumper()`, and the impl function would now be named `pickler_dumper_impl()`.

Similarly, you may have a problem where you want to give a parameter a specific Python name, but that name may be inconvenient in C. Argument Clinic allows you to give a parameter different names in Python and in C, using the same "as" syntax:

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
    file as file_obj: object
    protocol: object = NULL
    *
    fix_imports: bool = True
```

Here, the name used in Python (in the signature and the `keywords` array) would be `file`, but the C variable would be named `file_obj`.

You can use this to rename the `self` parameter too!

4.3 PyArg_UnpackTuple による関数の変換

To convert a function parsing its arguments with `PyArg_UnpackTuple()`, simply write out all the arguments, specifying each as an `object`. You may specify the `type` argument to cast the type as appropriate. All arguments should be marked positional-only (add a `/` on a line by itself after the last argument).

Currently the generated code will use `PyArg_ParseTuple()`, but this will change soon.

4.4 オプション群

Some legacy functions have a tricky approach to parsing their arguments: they count the number of positional arguments, then use a `switch` statement to call one of several different `PyArg_ParseTuple()` calls depending on how many positional arguments there are. (These functions cannot accept keyword-only arguments.) This approach was used to simulate optional arguments back before `PyArg_ParseTupleAndKeywords()` was created.

While functions using this approach can often be converted to use `PyArg_ParseTupleAndKeywords()`, optional arguments, and default values, it's not always possible. Some of these legacy functions have behaviors `PyArg_ParseTupleAndKeywords()` doesn't directly support. The most obvious example is the builtin function `range()`, which has an optional argument on the *left* side of its required argument! Another example is `curses.window.addch()`, which has a group of two arguments that must always be specified together. (The arguments are called `x` and `y`; if you call the function passing in `x`, you must also pass in `y`—and if you don't pass in `x` you may not pass in `y` either.)

In any case, the goal of Argument Clinic is to support argument parsing for all existing CPython builtins without changing their semantics. Therefore Argument Clinic supports this alternate approach to parsing, using what are called *optional groups*. Optional groups are groups of arguments that must all be passed in together. They can be to the left or the right of the required arguments. They can *only* be used with positional-only parameters.

注释: Optional groups are *only* intended for use when converting functions that make multiple calls to `PyArg_ParseTuple()`! Functions that use *any* other approach for parsing arguments should *almost never* be converted to Argument Clinic using optional groups. Functions using optional groups currently cannot have accurate signatures in Python, because Python just doesn't understand the concept. Please avoid using optional groups wherever possible.

To specify an optional group, add a `[` on a line by itself before the parameters you wish to group together, and a `]` on a line by itself after these parameters. As an example, here's how `curses.window.addch` uses optional groups to make the first two parameters and the last parameter optional:

```
/*[clinic input]

curses.window.addch

    [
    x: int
      X-coordinate.
    y: int
      Y-coordinate.
    ]

    ch: object
      Character to add.

    [
    attr: long
      Attributes for the character.
    ]
    /

...
```

注释:

- For every optional group, one additional parameter will be passed into the impl function representing the group. The parameter will be an int named `group_{direction}_{number}`, where `{direction}` is either `right` or `left` depending on whether the group is before or after the required parameters, and `{number}` is a monotonically increasing number (starting at 1) indicating how far away the group is from the required parameters. When the impl is called, this parameter will be set to zero if this group was unused, and set to non-zero if this group was used. (By used or unused, I mean whether or not the parameters received arguments in this invocation.)
- If there are no required arguments, the optional groups will behave as if they're to the right of the required arguments.
- In the case of ambiguity, the argument parsing code favors parameters on the left (before the

required parameters).

- Optional groups can only contain positional-only parameters.
- Optional groups are *only* intended for legacy code. Please do not use optional groups for new code.

4.5 Using real Argument Clinic converters, instead of "legacy converters"

To save time, and to minimize how much you need to learn to achieve your first port to Argument Clinic, the walkthrough above tells you to use "legacy converters". "Legacy converters" are a convenience, designed explicitly to make porting existing code to Argument Clinic easier. And to be clear, their use is acceptable when porting code for Python 3.4.

However, in the long term we probably want all our blocks to use Argument Clinic's real syntax for converters. Why? A couple reasons:

- The proper converters are far easier to read and clearer in their intent.
- There are some format units that are unsupported as "legacy converters", because they require arguments, and the legacy converter syntax doesn't support specifying arguments.
- In the future we may have a new argument parsing library that isn't restricted to what `PyArg_ParseTuple()` supports; this flexibility won't be available to parameters using legacy converters.

Therefore, if you don't mind a little extra effort, please use the normal converters instead of legacy converters.

In a nutshell, the syntax for Argument Clinic (non-legacy) converters looks like a Python function call. However, if there are no explicit arguments to the function (all functions take their default values), you may omit the parentheses. Thus `bool` and `bool()` are exactly the same converters.

All arguments to Argument Clinic converters are keyword-only. All Argument Clinic converters accept the following arguments:

`c_default` The default value for this parameter when defined in C. Specifically, this will be the initializer for the variable declared in the "parse function". See *the section on default values* for how to use this. Specified as a string.

`annotation` The annotation value for this parameter. Not currently supported, because **PEP 8** mandates that the Python library may not use annotations.

In addition, some converters accept additional arguments. Here is a list of these arguments, along with their meanings:

`accept` A set of Python types (and possibly pseudo-types); this restricts the allowable Python argument to values of these types. (This is not a general-purpose facility; as a rule it only supports specific lists of types as shown in the legacy converter table.)

To accept None, add NoneType to this set.

bitwise Only supported for unsigned integers. The native integer value of this Python argument will be written to the parameter without any range checking, even for negative values.

converter Only supported by the object converter. Specifies the name of a C "converter function" to use to convert this object to a native type.

encoding Only supported for strings. Specifies the encoding to use when converting this string from a Python str (Unicode) value into a C char * value.

subclass_of Only supported for the object converter. Requires that the Python value be a subclass of a Python type, as expressed in C.

type Only supported for the object and self converters. Specifies the C type that will be used to declare the variable. Default value is "PyObject *".

zeroes Only supported for strings. If true, embedded NUL bytes ('\\0') are permitted inside the value. The length of the string will be passed in to the impl function, just after the string parameter, as a parameter named <parameter_name>_length.

Please note, not every possible combination of arguments will work. Usually these arguments are implemented by specific PyArg_ParseTuple *format units*, with specific behavior. For example, currently you cannot call unsigned_short without also specifying bitwise=True. Although it's perfectly reasonable to think this would work, these semantics don't map to any existing format unit. So Argument Clinic doesn't support it. (Or, at least, not yet.)

以下の表は、legacy converter から実際の Argument Clinic converter へのマッピングを示す表です。左側は legacy converter で、右側がそれを置き換えたテキストです。

'B'	unsigned_char(bitwise=True)
'b'	unsigned_char
'c'	char
'C'	int(accept={str})
'd'	double
'D'	Py_complex
'es'	str(encoding='name_of_encoding')
'es#'	str(encoding='name_of_encoding', zeroes=True)
'et'	str(encoding='name_of_encoding', accept={bytes, bytearray, str})
'et#'	str(encoding='name_of_encoding', accept={bytes, bytearray, str}, zeroes=True)
'f'	float
'h'	short
'H'	unsigned_short(bitwise=True)
'i'	int
'I'	unsigned_int(bitwise=True)
'k'	unsigned_long(bitwise=True)

次のページに続く

表 1 – 前のページからの続き

'K'	unsigned_long_long(bitwise=True)
'l'	long
'L'	long long
'n'	Py_ssize_t
'O'	object
'O!'	object(subclass_of='&PySomething_Type')
'O&'	object(converter='name_of_c_function')
'p'	bool
'S'	PyBytesObject
's'	str
's#'	str(zeroes=True)
's*'	Py_buffer(accept={buffer, str})
'U'	unicode
'u'	Py_UNICODE
'u#'	Py_UNICODE(zeroes=True)
'w*'	Py_buffer(accept={rwbuffer})
'Y'	PyByteArrayObject
'y'	str(accept={bytes})
'y#'	str(accept={robuffer}, zeroes=True)
'y*'	Py_buffer
'Z'	Py_UNICODE(accept={str, NoneType})
'Z#'	Py_UNICODE(accept={str, NoneType}, zeroes=True)
'z'	str(accept={str, NoneType})
'z#'	str(accept={str, NoneType}, zeroes=True)
'z*'	Py_buffer(accept={buffer, str, NoneType})

例題の `pickle.Pickler.dump` に適切な converter を使用したものを以下に示します:

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

One advantage of real converters is that they're more flexible than legacy converters. For example, the `unsigned_int` converter (and all the `unsigned_` converters) can be specified without `bitwise=True`. Their default behavior performs range checking on the value, and they won't accept negative numbers. You just can't do that with a legacy converter!

Argument Clinic will show you all the converters it has available. For each converter it'll show you all the parameters it accepts, along with the default value for each parameter. Just run `Tools/clinic/clinic.py --converters` to see the full list.

4.6 Py_buffer

When using the `Py_buffer` converter (or the `'s*'`, `'w*'`, `'*y'`, or `'z*'` legacy converters), you *must* not call `PyBuffer_Release()` on the provided buffer. Argument Clinic generates code that does it for you (in the parsing function).

4.7 Advanced converters

Remember those format units you skipped for your first time because they were advanced? Here's how to handle those too.

The trick is, all those format units take arguments—either conversion functions, or types, or strings specifying an encoding. (But "legacy converters" don't support arguments. That's why we skipped them for your first function.) The argument you specified to the format unit is now an argument to the converter; this argument is either `converter` (for `O&`), `subclass_of` (for `O!`), or `encoding` (for all the format units that start with `e`).

When using `subclass_of`, you may also want to use the other custom argument for `object()`: `type`, which lets you set the type actually used for the parameter. For example, if you want to ensure that the object is a subclass of `PyUnicode_Type`, you probably want to use the converter `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`.

One possible problem with using Argument Clinic: it takes away some possible flexibility for the format units starting with `e`. When writing a `PyArg_Parse` call by hand, you could theoretically decide at runtime what encoding string to pass in to `PyArg_ParseTuple()`. But now this string must be hard-coded at Argument-Clinic-preprocessing-time. This limitation is deliberate; it made supporting this format unit much easier, and may allow for future optimizations. This restriction doesn't seem unreasonable; CPython itself always passes in static hard-coded encoding strings for parameters whose format units start with `e`.

4.8 引数のデフォルト値

Default values for parameters can be any of a number of values. At their simplest, they can be string, int, or float literals:

```
foo: str = "abc"
bar: int = 123
bat: float = 45.6
```

They can also use any of Python's built-in constants:

```
yep:  bool = True
nope: bool = False
nada: object = None
```

There's also special support for a default value of NULL, and for simple expressions, documented in the following sections.

4.9 NULL デフォルト値

For string and object parameters, you can set them to None to indicate that there's no default. However, that means the C variable will be initialized to Py_None. For convenience's sakes, there's a special value called NULL for just this reason: from Python's perspective it behaves like a default value of None, but the C variable is initialized with NULL.

4.10 デフォルト値として指定された式

The default value for a parameter can be more than just a literal value. It can be an entire expression, using math operators and looking up attributes on objects. However, this support isn't exactly simple, because of some non-obvious semantics.

Consider the following example:

```
foo: Py_ssize_t = sys.maxsize - 1
```

sys.maxsize can have different values on different platforms. Therefore Argument Clinic can't simply evaluate that expression locally and hard-code it in C. So it stores the default in such a way that it will get evaluated at runtime, when the user asks for the function's signature.

What namespace is available when the expression is evaluated? It's evaluated in the context of the module the builtin came from. So, if your module has an attribute called "max_widgets", you may simply use it:

```
foo: Py_ssize_t = max_widgets
```

If the symbol isn't found in the current module, it fails over to looking in sys.modules. That's how it can find sys.maxsize for example. (Since you don't know in advance what modules the user will load into their interpreter, it's best to restrict yourself to modules that are preloaded by Python itself.)

Evaluating default values only at runtime means Argument Clinic can't compute the correct equivalent C default value. So you need to tell it explicitly. When you use an expression, you must also specify the equivalent expression in C, using the c_default parameter to the converter:

```
foo: Py_ssize_t(c_default="PY_SSIZE_T_MAX - 1") = sys.maxsize - 1
```

Another complication: Argument Clinic can't know in advance whether or not the expression you supply is valid. It parses it to make sure it looks legal, but it can't *actually* know. You must be very careful when using expressions to specify values that are guaranteed to be valid at runtime!

Finally, because expressions must be representable as static C values, there are many restrictions on legal expressions. Here's a list of Python features you're not permitted to use:

- 関数呼び出し
- インライン if 文 (`3 if foo else 5`).
- シークエンスの自動アンパック (`*[1, 2, 3]`).
- List/set/dict 内包表記とジェネレータ式
- Tuple/list/set/dict literals.

4.11 Using a return converter

By default the impl function Argument Clinic generates for you returns `PyObject *`. But your C function often computes some C type, then converts it into the `PyObject *` at the last moment. Argument Clinic handles converting your inputs from Python types into native C types—why not have it convert your return value from a native C type into a Python type too?

That's what a "return converter" does. It changes your impl function to return some C type, then adds code to the generated (non-impl) function to handle converting that value into the appropriate `PyObject *`.

The syntax for return converters is similar to that of parameter converters. You specify the return converter like it was a return annotation on the function itself. Return converters behave much the same as parameter converters; they take arguments, the arguments are all keyword-only, and if you're not changing any of the default arguments you can omit the parentheses.

(If you use both `"as"` *and* a return converter for your function, the `"as"` should come before the return converter.)

There's one additional complication when using return converters: how do you indicate an error has occurred? Normally, a function returns a valid (non-`NULL`) pointer for success, and `NULL` for failure. But if you use an integer return converter, all integers are valid. How can Argument Clinic detect an error? Its solution: each return converter implicitly looks for a special value that indicates an error. If you return that value, and an error has been set (`PyErr_Occurred()` returns a true value), then the generated code will propagate the error. Otherwise it will encode the value you return like normal.

Currently Argument Clinic supports only a few return converters:

```
bool
int
unsigned int
long
unsigned int
```

(次のページに続く)

```
size_t
Py_ssize_t
float
double
DecodeFSDefault
```

None of these take parameters. For the first three, return `-1` to indicate error. For `DecodeFSDefault`, the return type is `const char *`; return a `NULL` pointer to indicate an error.

(There's also an experimental `NoneType` converter, which lets you return `Py_None` on success or `NULL` on failure, without having to increment the reference count on `Py_None`. I'm not sure it adds enough clarity to be worth using.)

To see all the return converters Argument Clinic supports, along with their parameters (if any), just run `Tools/clinic/clinic.py --converters` for the full list.

4.12 既存関数の複製

If you have a number of functions that look similar, you may be able to use Clinic's "clone" feature. When you clone an existing function, you reuse:

- its parameters, including
  - 名前、
  - コンバータと全引数、
  - デフォルト値、
  - 引数ごとのドックストリング、
  - **種類** (位置専用、位置またはキーワード、キーワード専用)、
- return コンバータ

The only thing not copied from the original function is its docstring; the syntax allows you to specify a new docstring.

Here's the syntax for cloning a function:

```
/*[clinic input]
module.class.new_function [as c_basename] = module.class.existing_function

Docstring for new_function goes here.
[clinic start generated code]*/
```

(The functions can be in different modules or classes. I wrote `module.class` in the sample just to illustrate that you must use the full path to *both* functions.)

Sorry, there's no syntax for partially cloning a function, or cloning a function then modifying it. Cloning is an all-or nothing proposition.

Also, the function you are cloning from must have been previously defined in the current file.

4.13 Python コードの呼び出し

The rest of the advanced topics require you to write Python code which lives inside your C file and modifies Argument Clinic's runtime state. This is simple: you simply define a Python block.

A Python block uses different delimiter lines than an Argument Clinic function block. It looks like this:

```
/*[python input]
# python code goes here
[python start generated code]*/
```

All the code inside the Python block is executed at the time it's parsed. All text written to stdout inside the block is redirected into the "output" after the block.

As an example, here's a Python block that adds a static integer variable to the C code:

```
/*[python input]
print('static int __ignored_unused_variable__ = 0;')
[python start generated code]*/
static int __ignored_unused_variable__ = 0;
/*[python checksum:...]*/
```

4.14 "self converter" の利用

Argument Clinic automatically adds a "self" parameter for you using a default converter. It automatically sets the `type` of this parameter to the "pointer to an instance" you specified when you declared the type. However, you can override Argument Clinic's converter and specify one yourself. Just add your own `self` parameter as the first parameter in a block, and ensure that its converter is an instance of `self_converter` or a subclass thereof.

What's the point? This lets you override the type of `self`, or give it a different default name.

How do you specify the custom type you want to cast `self` to? If you only have one or two functions with the same type for `self`, you can directly use Argument Clinic's existing `self` converter, passing in the type you want to use as the `type` parameter:

```
/*[clinic input]

_pickle.Pickler.dump

  self: self(type="PicklerObject *")
  obj: object
```

(次のページに続く)

(前のページからの続き)

```
  /

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

On the other hand, if you have a lot of functions that will use the same type for `self`, it's best to create your own converter, subclassing `self_converter` but overwriting the `type` member:

```
/*[python input]
class PicklerObject_converter(self_converter):
    type = "PicklerObject *"
[python start generated code]*/

/*[clinic input]

_pickle.Pickler.dump

  self: PicklerObject
  obj: object
  /

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

4.15 Using a "defining class" converter

Argument Clinic facilitates gaining access to the defining class of a method. This is useful for heap type methods that need to fetch module level state. Use `PyType_FromModuleAndSpec()` to associate a new heap type with a module. You can now use `PyType_GetModuleState()` on the defining class to fetch the module state, for example from a module method.

Example from `Modules/zlibmodule.c`. First, `defining_class` is added to the clinic input:

```
/*[clinic input]
zlib.Compress.compress

  cls: defining_class
  data: Py_buffer
    Binary data to be compressed.
  /
```

After running the Argument Clinic tool, the following function signature is generated:

```
/*[clinic start generated code]*/
static PyObject *
zlib_Compress_compress_impl(compobject *self, PyTypeObject *cls,
                            Py_buffer *data)
/*[clinic end generated code: output=6731b3f0ff357ca6 input=04d00f65ab01d260]*/
```

The following code can now use `PyType_GetModuleState(cls)` to fetch the module state:

```
zlibstate *state = PyType_GetModuleState(cls);
```

Each method may only have one argument using this converter, and it must appear after `self`, or, if `self` is not used, as the first argument. The argument will be of type `PyTypeObject *`. The argument will not appear in the `__text_signature__`.

The `defining_class` converter is not compatible with `__init__` and `__new__` methods, which cannot use the `METH_METHOD` convention.

It is not possible to use `defining_class` with slot methods. In order to fetch the module state from such methods, use `_PyType_GetModuleByDef` to look up the module and then `PyModule_GetState()` to fetch the module state. Example from the `setattro` slot method in `Modules/_threadmodule.c`:

```
static int
local_setattro(localobject *self, PyObject *name, PyObject *v)
{
    PyObject *module = _PyType_GetModuleByDef(Py_TYPE(self), &thread_module);
    thread_module_state *state = get_thread_state(module);
    ...
}
```

See also **PEP 573**.

4.16 カスタムコンバータを書く

As we hinted at in the previous section... you can write your own converters! A converter is simply a Python class that inherits from `CConverter`. The main purpose of a custom converter is if you have a parameter using the `O&` format unit—parsing this parameter means calling a `PyArg_ParseTuple()` "converter function".

Your converter class should be named `*something*_converter`. If the name follows this convention, then your converter class will be automatically registered with Argument Clinic; its name will be the name of your class with the `_converter` suffix stripped off. (This is accomplished with a metaclass.)

You shouldn't subclass `CConverter.__init__`. Instead, you should write a `converter_init()` function. `converter_init()` always accepts a `self` parameter; after that, all additional parameters *must* be keyword-only. Any arguments passed in to the converter in Argument Clinic will be passed along to your `converter_init()`.

There are some additional members of `CConverter` you may wish to specify in your subclass. Here's the current list:

`type` The C type to use for this variable. `type` should be a Python string specifying the type, e.g. `int`. If this is a pointer type, the type string should end with `' *'`.

`default` The Python default value for this parameter, as a Python value. Or the magic value `unspecified` if there is no default.

`py_default` `default` as it should appear in Python code, as a string. Or `None` if there is no default.

`c_default` `default` as it should appear in C code, as a string. Or `None` if there is no default.

`c_ignored_default` The default value used to initialize the C variable when there is no default, but not specifying a default may result in an "uninitialized variable" warning. This can easily happen when using option groups—although properly written code will never actually use this value, the variable does get passed in to the impl, and the C compiler will complain about the "use" of the uninitialized value. This value should always be a non-empty string.

`converter` The name of the C converter function, as a string.

`impl_by_reference` A boolean value. If true, Argument Clinic will add a `&` in front of the name of the variable when passing it into the impl function.

`parse_by_reference` A boolean value. If true, Argument Clinic will add a `&` in front of the name of the variable when passing it into `PyArg_ParseTuple()`.

Here's the simplest example of a custom converter, from `Modules/zlibmodule.c`:

```
/*[python input]

class ssize_t_converter(CConverter):
    type = 'Py_ssize_t'
    converter = 'ssize_t_converter'

[python start generated code]*/
/*[python end generated code: output=da39a3ee5e6b4b0d input=35521e4e733823c7]*/
```

This block adds a converter to Argument Clinic named `ssize_t`. Parameters declared as `ssize_t` will be declared as type `Py_ssize_t`, and will be parsed by the `'O&'` format unit, which will call the `ssize_t_converter` converter function. `ssize_t` variables automatically support default values.

More sophisticated custom converters can insert custom C code to handle initialization and cleanup. You can see more examples of custom converters in the CPython source tree; grep the C files for the string `CConverter`.

4.17 カスタム return コンバータを書く

Writing a custom return converter is much like writing a custom converter. Except it's somewhat simpler, because return converters are themselves much simpler.

Return converters must subclass `CReturnConverter`. There are no examples yet of custom return converters, because they are not widely used yet. If you wish to write your own return converter, please read `Tools/clinic/clinic.py`, specifically the implementation of `CReturnConverter` and all its subclasses.

4.18 METH_O と METH_NOARGS

To convert a function using METH_O, make sure the function's single argument is using the object converter, and mark the arguments as positional-only:

```
/*[clinic input]
meth_o_sample

     argument: object
     /
[clinic start generated code]*/
```

To convert a function using METH_NOARGS, just don't specify any arguments.

You can still use a self converter, a return converter, and specify a type argument to the object converter for METH_O.

4.19 tp_new と tp_init functions

You can convert tp_new and tp_init functions. Just name them __new__ or __init__ as appropriate. Notes:

- The function name generated for __new__ doesn't end in __new__ like it would by default. It's just the name of the class, converted into a valid C identifier.
- No PyMethodDef #define is generated for these functions.
- __init__ functions return int, not PyObject *.
- Use the docstring as the class docstring.
- Although __new__ and __init__ functions must always accept both the args and kwargs objects, when converting you may specify any signature for these functions that you like. (If your function doesn't support keywords, the parsing function generated will throw an exception if it receives any.)

4.20 Clinic 出力の変更とリダイレクト

It can be inconvenient to have Clinic's output interspersed with your conventional hand-edited C code. Luckily, Clinic is configurable: you can buffer up its output for printing later (or earlier!), or write its output to a separate file. You can also add a prefix or suffix to every line of Clinic's generated output.

While changing Clinic's output in this manner can be a boon to readability, it may result in Clinic code using types before they are defined, or your code attempting to use Clinic-generated code before it is defined. These problems can be easily solved by rearranging the declarations in your file, or moving where Clinic's generated code goes. (This is why the default behavior of Clinic is to output everything into the current block; while many people consider this hampers readability, it will never require rearranging your code to fix definition-before-use problems.)

Let's start with defining some terminology:

field A field, in this context, is a subsection of Clinic's output. For example, the `#define` for the `PyMethodDef` structure is a field, called `methoddef_define`. Clinic has seven different fields it can output per function definition:

```
docstring_prototype
docstring_definition
methoddef_define
impl_prototype
parser_prototype
parser_definition
impl_definition
```

All the names are of the form `"<a>_<b>"`, where `"<a>"` is the semantic object represented (the parsing function, the impl function, the docstring, or the methoddef structure) and `"<b>"` represents what kind of statement the field is. Field names that end in `"_prototype"` represent forward declarations of that thing, without the actual body/data of the thing; field names that end in `"_definition"` represent the actual definition of the thing, with the body/data of the thing. (`"methoddef"` is special, it's the only one that ends with `"_define"`, representing that it's a preprocessor #define.)

destination A destination is a place Clinic can write output to. There are five built-in destinations:

`block` The default destination: printed in the output section of the current Clinic block.

`buffer` A text buffer where you can save text for later. Text sent here is appended to the end of any existing text. It's an error to have any text left in the buffer when Clinic finishes processing a file.

`file` A separate "clinic file" that will be created automatically by Clinic. The filename chosen for the file is `{basename}.clinic{extension}`, where `basename` and `extension` were assigned the output from `os.path.splitext()` run on the current file. (Example: the `file` destination for `_pickle.c` would be written to `_pickle.clinic.c`.)

Important: When using a `file` destination, you *must check in* the generated file!

`two-pass` A buffer like `buffer`. However, a two-pass buffer can only be dumped once, and it prints out all text sent to it during all processing, even from Clinic blocks *after* the dumping point.

`suppress` The text is suppressed—thrown away.

Clinic defines five new directives that let you reconfigure its output.

The first new directive is `dump`:

```
dump <destination>
```

This dumps the current contents of the named destination into the output of the current block, and empties it. This only works with `buffer` and `two-pass` destinations.

The second new directive is `output`. The most basic form of `output` is like this:

```
output <field> <destination>
```

This tells Clinic to output *field* to *destination*. `output` also supports a special meta-destination, called `everything`, which tells Clinic to output *all* fields to that *destination*.

`output` has a number of other functions:

```
output push
output pop
output preset <preset>
```

`output push` and `output pop` allow you to push and pop configurations on an internal configuration stack, so that you can temporarily modify the output configuration, then easily restore the previous configuration. Simply push before your change to save the current configuration, then pop when you wish to restore the previous configuration.

`output preset` sets Clinic's output to one of several built-in preset configurations, as follows:

`block` Clinic's original starting configuration. Writes everything immediately after the input block.

Suppress the `parser_prototype` and `docstring_prototype`, write everything else to `block`.

`file` Designed to write everything to the "clinic file" that it can. You then `#include` this file near the top of your file. You may need to rearrange your file to make this work, though usually this just means creating forward declarations for various `typedef` and `PyTypeObject` definitions.

Suppress the `parser_prototype` and `docstring_prototype`, write the `impl_definition` to `block`, and write everything else to `file`.

The default filename is `"{dirname}/clinic/{basename}.h"`.

`buffer` Save up most of the output from Clinic, to be written into your file near the end. For Python files implementing modules or builtin types, it's recommended that you dump the buffer just above the static structures for your module or builtin type; these are normally very near the end. Using `buffer` may require even more editing than `file`, if your file has static `PyMethodDef` arrays defined in the middle of the file.

Suppress the `parser_prototype`, `impl_prototype`, and `docstring_prototype`, write the `impl_definition` to `block`, and write everything else to `file`.

`two-pass` Similar to the `buffer` preset, but writes forward declarations to the `two-pass` buffer, and definitions to the `buffer`. This is similar to the `buffer` preset, but may require less editing than `buffer`. Dump the `two-pass` buffer near the top of your file, and dump the `buffer` near the end just like you would when using the `buffer` preset.

Suppresses the `impl_prototype`, write the `impl_definition` to `block`, write `docstring_prototype`, `methoddef_define`, and `parser_prototype` to `two-pass`, write everything else to `buffer`.

`partial-buffer` Similar to the `buffer` preset, but writes more things to `block`, only writing the really big chunks of generated code to `buffer`. This avoids the definition-before-use problem of `buffer` completely, at the small cost of having slightly more stuff in the block's output. Dump the `buffer` near the end, just like you would when using the `buffer` preset.

Suppresses the `impl_prototype`, write the `docstring_definition` and `parser_definition` to `buffer`, write everything else to `block`.

The third new directive is `destination`:

```
destination <name> <command> [...]
```

This performs an operation on the destination named `name`.

There are two defined subcommands: `new` and `clear`.

The `new` subcommand works like this:

```
destination <name> new <type>
```

This creates a new destination with name `<name>` and type `<type>`.

There are five destination types:

`suppress` Throws the text away.

`block` Writes the text to the current block. This is what Clinic originally did.

`buffer` A simple text buffer, like the ”buffer” builtin destination above.

`file` A text file. The file destination takes an extra argument, a template to use for building the filename, like so:

destination <name> new <type> <file_template>

The template can use three strings internally that will be replaced by bits of the filename:

{path} The full path to the file, including directory and full filename.

{dirname} The name of the directory the file is in.

{basename} Just the name of the file, not including the directory.

{basename_root} Basename with the extension clipped off (everything up to but not including the last '.').

{basename_extension} The last '.' and everything after it. If the basename does not contain a period, this will be the empty string.

If there are no periods in the filename, {basename} and {filename} are the same, and {extension} is empty. "{basename}{extension}" is always exactly the same as "{filename}"."

`two-pass` A two-pass buffer, like the "two-pass" builtin destination above.

The `clear` subcommand works like this:

```
destination <name> clear
```

It removes all the accumulated text up to this point in the destination. (I don't know what you'd need this for, but I thought maybe it'd be useful while someone's experimenting.)

The fourth new directive is `set`:

```
set line_prefix "string"
set line_suffix "string"
```

`set` lets you set two internal variables in Clinic. `line_prefix` is a string that will be prepended to every line of Clinic's output; `line_suffix` is a string that will be appended to every line of Clinic's output.

以下はいずれも書式文字列をサポートします:

`{block comment start}` Turns into the string `/*`, the start-comment text sequence for C files.

`{block comment end}` Turns into the string `*/`, the end-comment text sequence for C files.

The final new directive is one you shouldn't need to use directly, called `preserve`:

```
preserve
```

This tells Clinic that the current contents of the output should be kept, unmodified. This is used internally by Clinic when dumping output into `file` files; wrapping it in a Clinic block lets Clinic use its existing checksum functionality to ensure the file was not modified by hand before it gets overwritten.

4.21 #ifdef トリック

If you're converting a function that isn't available on all platforms, there's a trick you can use to make life a little easier. The existing code probably looks like this:

```
#ifdef HAVE_FUNCTIONNAME
static module_functionname(...)
{
...
```

(次のページに続く)

```
}
#endif /* HAVE_FUNCTIONNAME */
```

And then in the PyMethodDef structure at the bottom the existing code will have:

```
#ifdef HAVE_FUNCTIONNAME
{'functionname', ... },
#endif /* HAVE_FUNCTIONNAME */
```

In this scenario, you should enclose the body of your impl function inside the #ifdef, like so:

```
#ifdef HAVE_FUNCTIONNAME
/*[clinic input]
module.functionname
...
[clinic start generated code]*/
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

Then, remove those three lines from the PyMethodDef structure, replacing them with the macro Argument Clinic generated:

```
MODULE_FUNCTIONNAME_METHODDEF
```

(You can find the real name for this macro inside the generated code. Or you can calculate it yourself: it's the name of your function as defined on the first line of your block, but with periods changed to underscores, uppercased, and "_METHODDEF" added to the end.)

Perhaps you're wondering: what if HAVE_FUNCTIONNAME isn't defined? The MODULE_FUNCTIONNAME_METHODDEF macro won't be defined either!

Here's where Argument Clinic gets very clever. It actually detects that the Argument Clinic block might be deactivated by the #ifdef. When that happens, it generates a little extra code that looks like this:

```
#ifndef MODULE_FUNCTIONNAME_METHODDEF
    #define MODULE_FUNCTIONNAME_METHODDEF
#endif /* !defined(MODULE_FUNCTIONNAME_METHODDEF) */
```

That means the macro always works. If the function is defined, this turns into the correct structure, including the trailing comma. If the function is undefined, this turns into nothing.

However, this causes one ticklish problem: where should Argument Clinic put this extra code when using the "block" output preset? It can't go in the output block, because that could be deactivated by the #ifdef. (That's the whole point!)

In this situation, Argument Clinic writes the extra code to the "buffer" destination. This may mean that you get a complaint from Argument Clinic:

```
Warning in file "Modules/posixmodule.c" on line 12357:
Destination buffer 'buffer' not empty at end of file, emptying.
```

When this happens, just open your file, find the `dump buffer` block that Argument Clinic added to your file (it'll be at the very bottom), then move it above the `PyMethodDef` structure where that macro is used.

4.22 Python ファイル内での Argument Clinic の利用

It's actually possible to use Argument Clinic to preprocess Python files. There's no point to using Argument Clinic blocks, of course, as the output wouldn't make any sense to the Python interpreter. But using Argument Clinic to run Python blocks lets you use Python as a Python preprocessor!

Since Python comments are different from C comments, Argument Clinic blocks embedded in Python files look slightly different. They look like this:

```
#/*[python input]
#print("def foo(): pass")
#[python start generated code]*/
def foo(): pass
#/*[python checksum:...]*/
```

索引